C-terp

Version 2.0
Lattice Variant

GIMPEL SOFTWARE

Reference Manual

for

C-terp

An Interpretive
Development Environment
for

C

Version 2.00
Lattice Variant

May, 1985

Gimpel Software
3207 Hogarth Lane
Collegville, Pa. 19426
(215) 584-4261

Disclaimer

Gimpel Software has taken due care in preparing this manual and the programs and data on the electronic disk media (if any) accompanying this book including research, development and testing to ascertain their effectiveness.

Gimpel Software makes no warranties as to the contents of this manual and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Gimpel Software further reserves the right to make changes to the specifications of the program and contents of the manual without obligation to notify any person or organization of such changes.

Trademarks

C-terp is a Trademark of Gimpel Software.
Lattice is a Trademark of Lattice, Inc.
C86 is a Trademark of Computer Innovations, Inc.
IBM is a Trademark of International Business Machines.
MS-DOS is a Trademark of Microsoft, Inc.
Unix is a Trademark of Bell Telephone Laboratories.

CONTENTS

BASIC FACTS

C-terp is a software package that operates on an IBM PC (or compatible machine) under MS-DOS 2.0 and up. It can operate in 192Kb of memory and will use all the memory at its disposal. C-terp interprets full K&R C. It can be linked via the IBM linker to C-written and assembler-written functions and data following the C86 or the Lattice C calling conventions.

INTRODUCTION

About C

C was designed and developed by Dennis Ritchie at Bell Laboratories in the early 1970's. Its primary purpose was to write "system programs." These are programs that need to be very efficient because they are used frequently by many individuals. The alternative to writing in a system implementation language is to write in assembly language. Assembly languages have the unfortunate properties of being difficult to write in and to understand, and are machine dependent. As a competitor to assembly language, C needed to incorporate many facilities we regard as being in the domain of machines rather than problem solvers. It includes bytes, pointers, a variety of integers (short, regular, long, unsigned) as well as the more traditional floating point data. C's operators also reflect facilities common in many machines such as bit-shifting, AND'ing, OR'ing, complementation, auto-incrementing, etc., in addition to the conventional arithmetic facilities.

The basic philosophy of C's design, then, was not to provide facilities that would be convenient to the programmer but to present the facilities that are normally present in machines in a clean, comprehensive and machine-independent manner.

Incidentally, the principal ancestor of C was Martin Richard's BCPL, a system programming language that had only one type, the word. This language was fine before the modern machine adopted the byte as the fundamental addressable unit.

For the Novice

If you do not already know C, this manual may prove awkward reading in places (although you should be able to read the GETTING STARTED section and enter the small program described there). There are a number of good books on C and some of these are listed in the section entitled RECOMMENDED READING. The definitive text for C is the book by Kernighan and Ritchie. You would do well to obtain this book and perhaps one other. Your learning will be greatly enhanced by typing in the examples found in K&R and observing their behavior, especially under one or more of the various trace modes.

About C-terp

C is a relatively rich language. Although it can be translated into an efficient program, this process usually takes time. The time is not so long if you are skilled enough to get the program and the effect you are trying to achieve right the first time. But if you are like most people you will make repeated compilations for such purposes as the insertion of debug statements, making slight modifications, attempting different strategies, etc.

I am normally a calm and reflective person willing to accept delays in a passive and philosophic manner, but when I am sitting before a machine which is seemingly endlessly compiling a program that I had modified by the insertion of a print statement just to find out what went wrong with the last compile, knowing full well I'm going to have to recompile again with the print statement pulled out... for a program that should have been done yesterday... I feel, as Dick Button (Olympic ice skating champion turned announcer) would say, "the knife turn."

This is where C-terp comes in. It is an interpreter or, more accurately, a semi-compiler. It translates the source program into a sequence of tokens in one pass at high speed. It remains resident while the tokens are interpreted to execute the program. C-terp comes with its own editor so that you never have to leave C-terp while passing from editing to compiling to running.

C-terp is designed to be compatible with your regular compiler so that the program that successfully interprets under C-terp will be compilable into a program that runs at high speed. You then have the best of both worlds.

There are many side advantages to fast compilation. While entering a program you can "press a button" (actually two keystrokes) and get compiler diagnostics virtually immediately. This gives you the effect of incremental compilation.

You can develop programs from the top down. For example, you can enter in just enough of your program to process the command line attending to details such as error messages for invalid or missing arguments, etc., and debug all this before passing on to the next phase, while the above considerations are fresh in your mind.

C-terp is especially helpful with programs that are multiple modules (more than one .c file). It provides a Global Search facility that looks through all files for a string pattern. It also contains an automatic recompile facility. When the Run command is invoked all .c modules that have been modified since they were last compiled or than include files than have been modified will be recompiled.

All in all, we feel that C-terp will be a valuable tool to enhance your productivity, your programs, and your peace of mind. If it falls short of these goals, please let us know.

GETTING STARTED

The Distribution Diskette

C-terp is distributed on one or more distribution diskettes. Before you accidentally destroy the information recorded there, it is recommended that you make a copy, save the original, and use just the copy.

Your distribution diskettes contain the following files:

read.me
install.exe

ct.exe
stdio.h

tblxn.c
tblxn.h
cterp.lib
ctlib.lib
c.obj
c.asm

In addition to these files there are a number of example programs whose names are of the form 'example#.c'. Instructions on how to run these are given at the end of this section.

The file read.me contains information of a timely up-to-date nature that may include corrections and/or updates to this manual. The file install.exe is a program that you can run to customize C-terp. See "The install program" below.

The next two files are files that you are likely to use in your everyday programming. The remaining files can be used to prepare a new version of C-terp containing added facilities. Their use will be explained in the section entitled "LINKING TO EXTERNALS".

The install program

You normally won't need the install program. It's purpose is to modify C-terp to change display attributes (i.e., the various special attributes employed to enhance or distinguish portions of the screen). You will need to use it if you are using a graphics board with a monochrome screen. You may use it if you are unhappy with the current selection of display attributes.

The name of the install program is install.exe. Run this program by typing install at command level and follow its directions. It will ask you for the names of files to modify. For these supply either or both of ct.exe and cterp.lib.

Note that any file so modified may be remodified. You may also revert back to the initial configuration by using an attribute of 00.

Editing your first program

Copy ct.exe and stdio.h into a convenient directory and invoke ct by typing:

ct

It is possible to invoke ct with an argument and this is discussed under BATCH MODE.

As a result of the ct command, the main menu will appear before your eyes. Here, there are a number of choices of things to do, and these are described in the section entitled "MAIN MENU". For the purpose of getting started we should focus in on just one task. We will create and run a very common first program, one that prints the simple message "Hello World".

One of the commands in the main menu will be "Edit". Type 'E' (or 'e') and you will be prompted for a file name. Assuming there is no file named "hello.c" in your current directory (if there is, pick a name other than "hello") type the word "hello" without the quotes and follow this with Enter. The screen should blank and there should appear a number 1 in the upper left hand corner, the name "hello.c" in the lower right and the words "new file" in the lower left. You may now start typing the program. Type:

```
main()
    {
    printf( "Hello World\n" );
    }
```

If you make a mistake you can try to correct it with the various editing keys on your keyboard. Many of the simple editing keys are obvious but a prominent one that isn't is the Ctrl Backspace which deletes a line. If you get stuck, you may obtain a help screen by typing Alt H. In general, this is a "what you see is what you get" editor and, at least for a small program, editing should be relatively straightforward with few surprises.

Running your first program

Once the program looks right, you may return to the main menu by pressing the Escape key. At the sight of the main menu, type 'r' for run. We could have telescoped these two keys into one by typing F2.

When you type 'r' for run, C-terp will automatically compile all .c files that have not been compiled since they were last edited. This certainly includes hello.c since we have just entered it.

If you haven't been following this script religiously but have decided to play around and load a number of other files, then any that have the .c extension will also be compiled; morever, if any of these have a main() entry point you will get a diagnostic. Your best hope then is to unload the excess baggage with the Unload command.

But assuming you've been following directions, you should now get a command line prompt. In this case there is nothing we want to place on the command line so just type Enter. Your program will run and the phrase "Hello World"

should appear on the screen. You will be invited to press a key and the main menu will again appear before you.

At this point you may reenter the editor by typing an 'e'. If you do you will be prompted for a file but this time you will be given a default file name within square brackets. The default is used if you just hit Enter. If you change your mind and decide you don't want to edit after all just type escape (Esc) sometime before hitting Enter.

If you do enter the editor you may modify and rerun the program using a repetition of the sequence described earlier.

Before you quit you might want to save the file on disk so that it can be used in subsequent sessions. Do this with the Write command following the same default-file convention as with other commands.

To quit, get to the main menu and type 'q'.

Running the examples

To run an example program it is not necessary to enter the editor. For this we may just load the file and issue a Run. Thus the steps to run example1.c are as follows: (Assume that example1.c is on drive B:)

Prompt	You Type:	Comment
System Prompt	ct <Enter>	Enter C-terp
Main Menu	l	Load ...
file name:	b:example1 <Enter>	the desired file
Main Menu	r	Run it
command line:	<Enter>	doesn't require arguments

Before running other examples, read the comments in the beginning of the file to determine what sort of command line is expected.

Using the large memory model

Many users of C-terp have never before used the large memory model. Their programs operate using the small model; will they work using the large? Then, once having gotten them to work in the large (under C-terp) will they then operate successfully in the small model?

The answer, happily, is that the differences are largely transparent. The most frequent cause of incompatibility in going to the large model is the failure to declare everything properly. To cite perhaps the most common case, a function f() is defined in a module to return a pointer and is referenced in another module (another file) but not declared (and hence defaulting to int). C-terp will catch this and other inter-module incompatibilities and hence there should be little, if any, grief spent on migrating to the large model. The added declarations will, of course, in no way adversely affect the program should it be compiled under the small model later on.

This is, of course, as it should be. The programmer should not have to know what model he/she is working in. There is one area where there is a real

difference, however. If you are using a software interrupt to obtain a service from the operating system (see bdos, int86x) and if you are attempting to pass a pointer to DOS, there is a big difference between the two models. Make sure you have such code protected with the appropriate pre-processor flags (see THE PREPROCESSOR).

THE MAIN MENU

The main menu contains the following commands

Compile	Pre-process
Edit	Quit
File list	Run
Global search	Unload
Load	Write

Each is triggered by typing an upper or lower case letter corresponding to the first letter of the command. Except for the Quit and File list commands you will be prompted for an argument. You may specify an appropriate argument followed by Enter. If the prompt contains information in square brackets such as:

file name [hello.c] :

then typing Enter will have the effect of having typed the information in square brackets (in this example "hello.c").

IMPORTANT: You may abort the command by typing escape (Esc) before the Enter. In general, whenever C-terp is prompting you for a line and you want to 'get out' type escape. Also, by pressing F8 you may rotate among all the files currently loaded.

File Names -- You may use full path names to specify files. If the file contains no dot-extension a .c is assumed. To specify a file that has no extension use an explicit dot. Thus "temp" implies "temp.c" whereas "temp." implies the file with no extension.

Compile

This command will prompt you for a module name; the named module will then be compiled. If the module had not previously been loaded it will be loaded into C-terp's memory first. In this case you will notice a short delay before compilation begins.

Compiling is distinguished by a display of the file name and the current line number. At termination of compilation the message "compilation complete" is displayed and you are back in the main menu. Otherwise a diagnostic is displayed and at the touch of a key you will be placed in the editor at the appropriate place.

Since compilation is automatically invoked with a Run command, why use the Compile command? One reason is to get fast feedback of syntactic errors

while entering a long program.

Edit

This command will prompt for a file name and enter the editor. If the file is not in main memory, there will be an attempt to load it. If the file does not exist on disk, you will be placed in the editor with a "new file" indication.

File list

To obtain a list of all the files currently loaded use this command. This list will contain all the include files (files indicated on a #include compiler directive) of any module that has been compiled and so it may contain some surprises.

Global search

You will be prompted for a pattern. The pattern is a string that will be matched against each position in each line of each file currently loaded. Every line containing the string will be printed (including the line number). The file name will be printed but only if it contains at least one match.

Load

To load a file means to bring it into memory so that C-terp knows about it. The sequence (Load, file-name, Run) is very common because once C-terp knows about a module (a file with an extension of .c) it will include it in any Run that is requested. Unload will reverse the effects of a Load.

Pre-process

The Pre-process command will invoke just the pre-processor. It is NOT needed to compile or run a program. Its purpose is to provide an indication of how the pre-processor is translating the program.

You will be prompted first for the input file and then the output file. The output file can subsequently be examined via the Edit command.

But beware if you are repeatedly using this command followed by an Edit. Consider the following scenario. Pre-process file.c outputting to file.p; Edit file.p; this works fine. Again, Pre-process file.c outputting to file.p; Edit file.p; not so fine. The request to Edit file.p will result in editing the first file.p not the one newly produced. The sequence should be modified so that file.p is Unloaded any time prior to the second Edit.

This caution should be borne in mind whenever you are repeatedly examining the output of some program, such as one being interpreted.

Quit

This causes C-terp to exit to system level. You will be prompted for each file modified but not yet written. If you type anything other than 'y' or 'Y' to this prompt, the file will not be saved. You must follow the 'y' with Enter.

Run

The purpose of the Run command is to send the program into execution. It will automatically compile each module (.c file) that has been modified since it was last compiled or that includes a file to any depth that has been modified since the module's last compilation. Note: if a .c file is included in another file via the #include facility it will lose its module status and will not actually be compiled.

The modules will then be linked together via their external definitions and references and, if any unresolved references remain after this process, the list of active external names will be searched. This list of external names is kept in a module called tblxn.c and procedures for modifying the table are described in the section titled "LINKING TO EXTERNALS". If there are still unresolved references after this process then the interpreter will resolve some special functions which include exit() and trace(). Finally, if these do not resolve the reference, a diagnostic is issued.

If the linking process generates errors, the user will be given a chance to correct them. The global search command will come in handy to locate names that are multiply-defined or have been left undefined.

Linking is very fast owing to the fact that everything is really pre-loaded. After a successful link, you are prompted for a command line. This gives you a last chance to 'escape out' of the run. The command line is given in a manner similar to the way it would be given at the DOS level. For example, if you are writing a program called examine which expects one argument and writes to standard out, you might type as a command:

```
e  file.in  >file.out
```

Here, we have abbreviated the name 'examine' to a single letter because the name itself is not significant in the program being run.

In the main program:

```
main(argc, argv)
int argc;
char **argv;
.
.
.
```

we would have argc equal to 2 with argv[0] equal to "e" and argv[1] equal to "file.in". file.out will have been opened as standard out but does not add to the argument count.

In general, operands are separated by blanks and tabs, and the redirection operators < and > are supported. It is all done in such a way that strong compatibility is maintained between the program being interpreted and the program as it would ultimately be compiled and executed.

At program termination, all open files are closed and all storage allocated via

alloc(), calloc(), malloc() and realloc() is released.

Unload

Unload will remove the file from main memory and C-terp will forget about its existence. This might be done to free up main memory or to prevent a .c module from being compiled into a run. Unload will not write out the file but will ask you if you want to save the file if it is one you have modified since having last written it out.

Write

This command will write the named file out to disk. For protection, the old file will be first renamed to filename.BAK. If the name provided is an asterisk ('*') all files modified since their last write will be written. This is useful before initiating a Run of an untested program.

If a Write Protect error message occurs you will be asked whether to "Abort, Retry or Ignore". This is probably due to a write-protected diskette. Remove the diskette, remove the write-protect tab, reinsert the disk and press 'R' for Retry.

THE EDITOR

The editor is intended to be a simple easy-to-use editor making maximum use of the special editing keys on the IBM keyboard. While it is oriented to the editing of program text it should be quite adequate to handle the editing of data as well.

Entering and Leaving the Editor

You enter the editor by typing 'e' at the main menu or by the system placing you in the editor as the result of an error of some kind. You may have as many files within the system as you want, space permitting.

You terminate editing a particular file by typing Escape or by typing one of three function keys:

Esc -- Quit to main menu

F1 -- Quit and Compile

F2 -- Quit and Run (implies compile of all ".c" files which have been modified or whose include files have been modified since the last compile)

F8 -- Next file (rotates among all files currently loaded)

Leaving the editor does not update the disk version of these files. This can be done via the Write command out of the main menu.

Most Obvious Editing Keys

Key	Effect
Ordinary character	character is inserted into text at the current cursor position and the cursor is moved right
Enter	inserts a blank line after the current line and automatically indents the cursor
Cursor-Left	moves the cursor left
Cursor-Right	moves the cursor right
Cursor-Up	moves the cursor up
Cursor-Down	moves the cursor down
Pg Up	displays preceding section of file
Pg Dn	displays next section of file
Del	deletes the character at the cursor
Backspace	deletes the character to the left of the cursor
Home	moves the cursor to the extreme left of the line
End	moves the cursor to the end of the line
Tab	moves the cursor to the next tab stop
Shift Tab	moves the cursor to the previous tab stop

The Ctrl Group

Commands in this group are invoked by depressing (and holding) the Ctrl key while striking one of the editing keys.

Ctrl Backspace	deletes the current line
Ctrl Home	positions the cursor at the beginning of the file
Ctrl End	positions the cursor at the end of the file
Ctrl Cursor-Left	moves the cursor one word to the left
Ctrl Cursor-Right	moves the cursor one word to the right
Ctrl PgUp	moves the cursor to the top of the screen
Ctrl PgDn	moves the cursor to the bottom of the screen

The Alt Group

The commands in this group are invoked by depressing and holding the Alt-mode key while striking a particular letter key. Some of these commands will prompt you for additional information. After you provide the information, type Enter. If you wish to abort the command (i.e., prevent it from carrying through to completion) type the escape key (Esc).

Alt C Copy -- You will be prompted for a line number. The default is the line the cursor is currently on. The command will copy the marked group of lines (see Alt K to find out how to marK lines) without disturbing them to a position just after the line designated. You may specify a line number of 0 in which case the lines are copied to the beginning of the current file. The marked range may be in some other file.

Alt G Global search -- You will be prompted for a string which you type followed by Enter. This will find and print all occurrences of the string in the file.

Alt H Help -- This command will provide a screenful of help information.

Alt I Interchange (Replace the locate-string with a new) -- You will be prompted for a string that is to replace the most recent locate-string. If the cursor is sitting just before such a locate string the interchange will take place without further prompt. In any event, it will go on and attempt to find a new instance of the locate-string and request you to type (Y/N/Q) for Yes (do the interchange), No (do not do the interchange but go on), or Quit. The default interchange string (if you type return when prompted for it) is the old interchange string. This is helpful when doing the same interchange over a series of files. If you want to replace the locate string with the null string enter Alt N.

Alt J Join -- will join the next line to the end of the current line. Leading white space is reduced to one space. See also the Split command (Alt S).

Alt K marK -- will mark the current line; the line number associated with the line will be highlighted (using reverse video). If a previous line had been marked, both lines will be taken as demarcating a range of lines. The set of marked lines may be copied or moved (see Alt C and Alt M). You may not have more than one range of lines marked at any one time; see Alt U. You may move a marked range from one file to another.

Alt L Locate -- You will be prompted for a string which you type followed by a return. This will locate the next occurrence of the string in the file. The search will wrap-around from the end of the file to the beginning of the file. (See Alt N.)

Alt M Move -- You will be prompted for a line number. The default is the line the cursor is currently on. The command will move the marked group of lines (see Alt K to find out how to marK lines) to a position just after the line designated. As with copy command (Alt C), the line number may be 0 and the marked lines may be in some other file. The original lines are deleted.

Alt N Next -- is a companion to Alt L. It will find the next occurrence of the string previously entered with Alt L.

Alt P Position -- You will be prompted for a line number. The cursor will be positioned to that line in the file.

Alt R Rename -- You will be prompted for a file name. The file will be relabeled to have this name. This command does not rename the corresponding file on disk but only changes the internal name used by C-terp. When a subsequent Write command is issued (from the main menu) it will write to the file on disk bearing this new name.

Alt S Split -- will split a line just before the current cursor into two lines. The new line is set to the same indentation as the old line. This is a companion to Join (Alt J).

Alt T Tabs -- You will be prompted for a number which will be used to reset the logical tab spacing (the initial value is 4). Logical tabs are spaced uniformly across the width of the line. See the section below on tabs.

Alt U Unmark -- will unmark the current marked range of lines allowing other lines to be marked.

Tabs

We distinguish between system tab settings and user (or logical) tab settings. System tabs occur every 8 positions and a tab within a file read from disk is assumed to be a system tab. A logical tab is the position to which the tab (or backtab) key will take you. Logical tabs are by default every four columns but may be changed by the Alt T command. When a line is edited or displayed, tab characters are converted to blanks according to the system tab settings. When the line is later stored leading blanks are converted to tab characters again according to system tab settings.

It is hoped in this way that we have the best of both worlds.

Long Lines

The longest line permitted is 255 characters. It may happen that a line is too long to fit on the screen. This normally will not cause a problem unless you wish to edit the portion of the line not on the screen. If this should occur, use the split command (Alt S) to decompose the line into something visible. Rejoin, if you must, with an Alt J.

THE PREPROCESSOR

Lines whose first character is a # are so-called preprocessor lines (See K&R, p. 207). The # must appear in column 1. You will get a diagnostic if the # is indented (this can easily happen with auto-indent editors). The preprocessor facilities described in K&R are all supported with the following additional considerations.

Pre-defined Symbols

C_terp

For C-terp, the pre-processor symbol C_terp is predefined. Thus:

```
#ifdef C_terp
#include "...stdio.h"
#else
#include "...stdio.h"
#endif
```

could be used to selectively read different header files depending on whether or not C-terp was being used. (Note that this normally is not necessary, see stdio.h below). The ellipsis, of course, could be a path name, a prefix character, or null, but would presumably be different for the two cases.

MSDOS I8086 I8086L LPTR

Each of these symbols is pre-defined to be 1 for compatibility with the Lattice C compiler under MS-DOS.

stdio.h

stdio.h is the Lattice stdio.h with the following change. If C-terp is processing stdio.h (#ifdef C_terp is true) getc and putc are #define'd as meaning fgetc and fputc respectively. Otherwise these two functions are #define'd as before. Thus, the C-terp stdio.h can also be used with your compiler and you need to bother with only one stdio.h. But the one stdio.h should be the one delivered with C-terp.

Incidentally, the only reason for not using the stdio.h in its original form is that the definitions of getc and putc call for modifying an area of storage beyond that allocated by the programmer. If check'ing were turned off (check(0)), the original getc and putc will work. This is not recommended, however.

Notes and Asides

#include "name"

and

#include <name>

are treated identically. name may be a full path name. Circularity is detected and inclusions may be nested up to a depth of 14.

#line <lineno>

is validated for correct syntax but otherwise has no effect. This pre-processor line is not meant for human eyes but is intended for communication between a pre-processor and pass 1 of the compiler.

DATA

Data declarations in C-terp are designed to mirror exactly the corresponding data definitions of a C compiler. This is so that data can be freely passed between routines operated interpretively and routines that have been compiled. The elemental objects are:

char

This object consumes one byte.

[unsigned] short [int]

In this implementation of C, a short int is identical to int. It occupies two bytes.

[unsigned] int

This object is a two-byte 8086 integer. If unsigned is present, appropriate unsigned variations of operations are performed.

long [int]

This object is a 4-byte quantity consisting of two int's (least signficant int first). The phrase "long int" is synonomous with "long".

pointer

Pointers are 4-byte quantities. The offset comes before the segment.

float

A float is 4 bytes long.

double or long float

A double is 8 bytes long. The format of floating point quantities is discussed in the section on LINKING TO EXTERNALS.

Alignment

Data in C-terp are aligned on a byte boundary. This is consistent with the Lattice compiler if the -b flag is used. Otherwise care must be taken as follows. Within structures in the Lattice C compiler, data is always aligned on a byte boundary. But arrays of structures are such that each structure within the array is aligned on a word boundary. To make arrays of structures accessible to both worlds, simply pad them out to an even byte length. This is good practice anyway so that access is not dependent on which flag was used. Indeed _iob in stdio.h is defined with a pad character.

DEBUGGING

C-terp offers two kinds of debugging aids, interactive debugging and tracing.

Interactive debugging

When the function breakpt() is executed or when one of a number of execution-time errors are encountered, C-terp enters into its interactive debug mode. This mode is recognizable by the portrayal of a section of the program in only the top portion of the screen. The bottom portion of the screen is devoted to entering and observing the results of C expressions. The menu at the middle of the screen describes the commands that can be entered.

Debugging mode is characterized by read-only editing, called browsing. While browsing, files may not be modified. However, all editing commands that do not change the contents of a file such as cursor movements and Page up, Page Dn, etc. may be employed. Also, you may use the F8 key to switch between files. The Escape, F1 and F2 keys will take you back to the main menu and will terminate the run.

You should see the following menu:

Display, Trace-back, Continue, Next-step, Edit, Restore, Window

To invoke any one of these commands you depress its initial letter (upper or lower case). Since the editor is in browse mode, typing a letter cannot be confused with wanting to insert a letter in the file. Indeed if you type any letter other than those that initiate a command you will receive a diagnostic.

D Display the value of a C expression. This allows you to evaluate any expression. See below for examples.

T Trace-back gives the sequence of function calls that brought control to this point.

C Continue on with the execution.

N Next step. This causes execution to continue for one more statement and you will automatically be placed back in debug mode.

E Edit the file (give up on this run). This allows you to start modifying the current file at the current position but execution is terminated. Any further execution must begin at the beginning.

R Restore will restore the editor state to the state it was in when the current breakpoint was first entered. This is handy if you've been roaming about in the editor and lost track of where control currently lies.

W Window will allow you to adjust the size of the viewing window (for the program). You will be prompted for a number.

An Example

Consider the following C program:

```
extern double sin();
int g;

main()
    {
    int n;
    n = 2;
    f();
    }

f()
    {
    int m;
    m = 3;
    breakpt();
    return m;
    }
```

Execution of the program will bring control to function f() where, after the assignment to m, breakpt() is executed. You will then be placed in debug mode. Then type d for display. You will be prompted for an expression. You may type any expression that can be evaluated at this point in the program. For example, any of the expressions shown below are valid in this context.

```
m
m + g
m = 15
m++
g * 2, m ? 3 : 2
f()
sin( 3.14 / 2 )
printf( "%x\n", m )
dumpv(1)
```

Note that the list does not include any expression containing variable n as

variable n is not normally accessible to an expression within function f(). Even if you manipulate cursor keys to place the cursor inside the main program you will still not be able to access variable n.

Note that some of the permitted expressions have side effects (of the above examples, m++ and y = 15 have side effects). Variables so modified retain their new value upon resuming execution.

As the examples suggest, you may make function calls upon built-in functions (dumpv(), which dumps variable values), external functions (printf() which you know about and sin() which you might add) and interpreted functions (such as f()). Thus, you may execute any function that you would normally be able to execute at this point in the program.

Calling function f() will trigger another breakpt(). The new breakpoint does not in fact wipe out the old. Were you to issue a Go command from the new breakpoint you will find yourself back at the old breakpoint with the value being displayed.

Note that functions (such as printf in the above examples) may be called even though not declared or referenced in the original program. The above program assumes that sin() is available as an external function (see the section on LINKING TO EXTERNALS). Such a function may be called and it will be assumed to return a double because of the declaration in the program. Were the declaration omitted, C-terp would assume that sin() returns an int.

One restriction: the pre-processor is not employed in interactive mode. This is because all #define's that were established when the module was compiled are long gone. Thus, putchar() could not normally be used as an expression to be entered interactively.

Breakpoints in C-terp are a little unusual in that triggering a breakpoint is accomplished by calling a function. This has many advantages however. For example, if we wanted to breakpoint whenever some function, say h(), was called with a negative value we could do so with:

```
h(n)
    int n;
    {
    if( n < 0 ) breakpt();
    .
    .
    .
```

Thus, you don't have to learn a new language to learn how to insert conditional breakpoints and you don't need a new language to learn how to set or display variables. On the other hand, breakpoints can't be set arbitrarily during a run. This is not normally a problem when compilation is fast.

Tracing

There is occasionally a desire to turn a switch that will produce relatively large quantities of output for relatively little effort such as the TRON statement of BASIC. C-terp offers several such forms of tracing.

Statement Tracing

Executing the function:

```
trace(1)
```

will turn on statement tracing. This will cause a message to be printed for each statement executed. The message will contain the line number at which the statment starts and the statement itself.

The function call:

```
trace(0)
```

will turn tracing back off.

Function Tracing

Executing the function:

```
trace(2)
```

will turn on function tracing. This will cause a message to be printed for each function call and for each return. The function call message will contain the name of the function as well as the values of the arguments. The function return message will contain the value returned. In order to properly associate call with return an indentation is made according to the level of function. The result is a striking display of function nesting levels.

As with statement tracing, trace(0) will turn function tracing off.

Types will be printed alongside the values if the number 4 is added to the argument provided to trace(). Thus,

```
trace(6)
```

will prepend a type designator onto all values printed during function tracing.

Assignments

```
trace( 010 )
```

will produce a trace for each assignment. If the left-hand-side of the assignment is a simple variable the variable's name will be printed.

Clause Expressions

```
trace( 020 )
```

will produce a trace for each expression in each clause. The clauses are the if(), while() and for() clauses.

Combinations

The arguments to trace() can be OR'ed to obtain a combined effect. Thus, trace(3) will initiate both statement and function tracing. The individual bits and their meanings are given below:

Bit 1	Statement Tracing
Bit 2	Function Tracing
Bit 4	Add type information to values printed
Bit 010	Assignment tracing
Bit 020	Clause tracing

BATCH MODE

On the command line invoking C-terp one may specify a file. If no extension is given, .ct is assumed. The file is loaded (as in the Load command). If the extension is not .ct no effort is made to interpret the file. If the extension is .ct, the lines of the file are interpreted as commands. For example if the command

```
ct batch.ct
```

is given and if batch.ct contains:

```
l temp
r temp temp.in >temp.out
q
```

then the l (load) command will cause temp.c to be loaded; the r (run) command will cause it to be compiled and executed; and the q (quit) command will cause C-terp to quit. In the case of the run command, the command line will be taken to be what follows the "r ".

In general, the commands in batch mode are the same as the commands in the main menu. The same defaults are also supplied. Thus, if the batch contains:

```
l prog1
r
u
l prog2
r
u
```

then the u (unload) command, though argumentless, will unload the current file

which will be the one previously loaded. Though these defaults are honored it may be safer and provide better documentation if the arguments are specified explicitly.

If the batch of commands does not contain a quit command, then at the completion of the batch, C-terp will drop into interactive mode. This is useful if you have a large number of files that comprise a C program.

For example, if alpha.ct contains:

```
l alpha1
l alpha2
l alpha3
l alpha4
l alpha5
```

then, if the programmer types:

```
ct  alpha
```

the files will be read in and the main menu will appear. At this point you are in interactive mode and may issue any of the usual commands.

THE LIBRARY

This section consists mostly of an alphabetic listing of the functions in the C-terp library accompanied by a detailed specification. The following 'tour' serves to group them into functional units and show more clearly how they relate to one another.

A Tour Through the Library

In 1966, the legendary Vic Vyssotsky was head of the Bell Labs effort in the Multics project (being done in cooperation with MIT and GE). The design of the I/O was floundering and Vic took over the job himself. Within a week a series of I/O documents (the BF. notes) emerged which changed the course of computing history. He regarded a file as a linear sequence of bytes totally abstracting away any hardware considerations such as sectors, tracks, physical records, record gaps, cylinders, etc. You could write n bytes, read n bytes, seek to any position, and that was about it. At the time, such simplicity was heresy and it took quite some time before the design was accepted by all parties to the Multics effort.

The Vyssotsky design was eventually adopted by Multics and by its very famous offspring, Unix. In fact, it contributed significantly to the basic Unix world-view (everything is a byte stream) and has been widely adopted in other systems. Although not present in DOS 1.x, it appears full blown in DOS 2.0 and beyond. We see it here in this I/O package and is largely provided for compatibility with the corresponding Unix calls. These are:

open	Open a file
creat	Create a file
close	Close a file
read	Read n bytes from a file
write	Write n bytes to a file
lseek	Seek to a position in the file

open returns a small integer that is used later to identify the file when reading, writing, seeking or closing. These are referred to as file descriptors.

There is one small difference between the DOS style byte stream and the Unix style. Unix uses a single character to separate lines (the newline) whereas DOS uses two (carriage-return/newline). Most applications written in C want to read and write just the newline. Hence under normal operation, on reading, the pair of characters is presented to the caller as a single newline and the reverse translation is made on a write. But it is possible to open a file in so-called binary mode in which case this translation is suppressed.

It was always intended that user-oriented facilities (such as read and write a number) would be built on top of these underlying facilities. The writers of such packages and utilities for the Unix operating system began to appreciate the importance of focusing the I/O down to atomic routines that would read or write a single byte. These were called getc and putc respectively. But if all I/O came down to calling these routines and if an operating system call were required for each character the overhead would be too great.

Toward the end of 1977 Dennis Ritchie (author of C) came up with a solution

that not only avoided frequent operating system calls but avoided, in most cases, the overhead of a function call. getc(p) would be a macro (as also would putc) which took as argument a pointer to a structure which either had sufficient characters in it, in which case one was extracted, or a call was made to fill the buffer. To implement this, files were designated by pointers and were declared as, for example:

```
FILE  *stream;
```

We use the word 'stream' in this documentation to distinguish from the small integer file descriptors mentioned earlier. This is nomenclature; both describe files and both are treated as a byte stream.

Because of the change of file designation, a whole new battery of functions corresponding to the earlier functions had to be created and these are:

fopen	Stream form of open
fclose	Stream form of close
fseek	Stream form of lseek
fread	Stream form of read
fwrite	Stream form of write

fopen returns a stream and each of the others requires a stream to designate the intended file.

The previously mentioned character operations are now as follows:

fgetc	Get a character (same as getc)
fputc	Put a character (same as putc)
getc	Get a character
getchar	Get a character from standard input
putc	Put a character
putchar	Put a character to standard output
ungetc	Push a character back onto a stream

The following functions deal with reading and writing lines to a stream.

fgets	Get a line (include newline)
gets	Get a line (strip the newline)
fputs	Put a string (companion to fgets)

Formatted I/O allows for conversions of numbers and general application oriented formatting.

fprintf	Formatted print to a file
fscanf	Formatted read from a file
printf	Formatted print to standard out
scanf	Formatted read from standard in
sprintf	Formatted print to a string
sscanf	Formatted read from a string

Finally, a number of additional I/O functions that all are based on the stream.

feof	End-of-file test

ferror	Error-on-file test
fflush	Flushes buffers
fileno	Return the underlying file descriptor

Storage allocation routines support the allocation of a fixed amount of storage and the ability to free exactly what was allocated.

calloc	Allocate and clear storage
free	Free storage
malloc	No frills memory allocation

Strings in C are null-terminated byte sequences. They are not particularly user-friendly (as in BASIC) but they are efficient and do not require a garbage collector.

strcat	Concatenate two strings
strcmp	Compare two strings
strcpy	Copy a string
strlen	Return the length of a string

But because strings are null-terminated additional functions are needed in cases where the null character is not where you want it or when it might lie where you don't want it.

movmem	Copy a sequence of bytes
setmem	Set an area of memory to a byte
strncpy	Copy up to n characters
strncmp	Compare up to n characters

The library contains a few conversions that can probably be done with sprintf or sscanf but not nearly as conveniently.

atof	Ascii (string) to float
atoi	Ascii (string) to integer
atoi	Ascii (string) to long
ftoa	Float to ascii (string)
ltos	Long to (ascii) string

Some functions test or translate characters.

isalpha	Is a character alphabetic?
isalnum	Is a character alphanumeric?
isdigit	Is a character a digit?
islower	Is a character lower case?
isspace	Is a character a space character?
isupper	Is a character upper case?
tolower	Convert to lower case.
toupper	Convert to upper case.

The following programs allow you to terminate execution.

exit	Terminate execution
i_exit	Terminate C-terp

There are several miscellaneous macros defined in stdio.h:

abs	absolute value
max	maximum of two numbers
min	minimum of two numbers
rewind	rewind a file

The following allow one to make operating system calls.

bdos	Execute DOS function
int86x	Execute an interrupt

Finally, there are a few functions that make direct requests upon C-terp.

trace	Start or stop tracing
check	Start or stop pointer checking
breakpt	Enter debug mode
dumpv	Dump variables

An Alphabetic Listing

The following list of functions are linked with C-terp as it is distributed. You may add to this list by linking in your favorite functions from your favorite libraries. See the section entitled "LINKING TO EXTERNALS."

abs -- Find the absolute value

Synopsis:

```
abs( n )
```

Description:

abs(n) will yield the absolute value of its argument. abs() is implemented as a macro (in stdio.h) and for this reason no type is shown for the argument and no type is shown for the return value. The advantage of implementing as a macro is that the function becomes truly generic working with floating point numbers equally as well as with integers. The disadvantage is that the argument should not have side-effects as it is referenced twice in the expansion.

atof -- Convert ASCII to floating point

Synopsis:

```
double atof(string)
    char *string;
```

Description:

atof() returns the floating point number represented by the string s. Initial white space in s is ignored. The string is scanned up to the first character not part of a floating point number. The number may appear with a sign and/or with a decimal point and/or with an exponent. The same conventions are used here as with fscanf.

atoi -- Convert ASCII to integer

Synopsis:

```
int  atoi(string)
       char *string;
```

Description:

atoi() returns an integer whose value is represented by the string given as argument. Initial white space is ignored. An optional sign should be followed by a sequence of digits. Scanning stops on the first non-digit. If no digits are there, 0 is returned.

atol -- Convert ASCII to long

Synopsis:

```
long  atol(string)
        char *string;
```

Description:

atol() returns a long integer whose value is represented by the string given as argument. Initial white space is ignored. An optional sign should be followed by a sequence of digits. Scanning stops on the first non-digit. If no digits are there, 0 is returned.

bdos -- Execute a DOS function

Synopsis:

```
int  bdos( fcode , d, al )
     int fcode;     /* function code */
     unsigned d;    /* DX value */
     unsigned al;   /* AL value */
```

Description:

This function executes INT 21H with AH set equal to fcode. The second argument, d, is assigned to DX before the call. The third argument is assigned to AL. The value returned is the contents of AL after the interrupt.

calloc -- Allocate and clear a block of memory

Synopsis:

```
char *calloc( nelem, elsize )
     unsigned nelem;
     unsigned elsize;
```

Description:

calloc() allocates nelem*elsize bytes of zeroed memory and returns a pointer to it. If that many bytes cannot be allocated, the NULL pointer is returned.

check -- Check pointers

Synopsis:

```
check(n)
     int n;
```

Description:

Pointers are normally checked for being within a reasonable range. check(0) will turn this off and check(1) will turn it back on again.

Pointer checking is done as follows: Anytime an assignment is executed, a check is made to be sure that the area assigned into is not lower than the least memory area available to the user nor higher than the highest memory area available. Memory is considered available if it is (a) part of static storage, (b) part of automatic storage of some activation record or (c) a portion of memory allocated via malloc, calloc or in general some routine dubbed as ALLOC in tblxn.c, or (d) a portion of memory allocated via a function dubbed as REALLOC in tblxn.c.

The form of a REALLOC function is f(ptr,int) returning ptr. Thus, if

you have an area of memory that you want accessible with check enabled, then you can mark a function in tblxn.c as REALLOC that does nothing but return its first argument.

Pointer checking is especially important in the big memory model because memory is otherwise unprotected and the operating system contains control information in lower memory (such as file allocation tables).

Pointer checking is, in general, not done by library functions. This is because most of the library functions accessed are the very same library functions you compiler uses. Thus, movmem, setmem, fprintf, etc. which logically could check pointers before acting, do not.

check(n) returns the most recent value passed to check(). This is helpful in a recursive environment.

close -- Close a file

Synopsis:

```
int close( fd )
    int fd;
```

Description:

close() will flush buffers and close the file for the file identified by fd (value returned from open() or creat()).

close() returns 0 if successful or -1 if an error was detected.

creat -- Create a new empty file

Synopsis:

```
int creat( filename, mode )
    char *filename;
    int mode;
```

Description:

This function may be used to create a new file but it offers no particular advantages over open() other than name compatibility with a similar function of Unix.

The file created will be the one designated by the first argument, filename. The allowable modes (second argument) are the same and have the same effect as the open call. The value returned is the same as that which would be returned by open().

dumpv -- Dump Variables

Synopsis:

```
dumpv( n )
    int n;
```

Description:

dumpv(n) will dump variables. dumpv(1) will dump local variables; dumpv(2) will dump global variables; dumpv(3) will dump local and global variables.

dumpv() is perhaps most useful in interactive debug mode.

exit -- Terminate a program

Synopsis:

```
exit( )
```

Description:

The program is terminated and, after a user keystroke, the main menu is regenerated. All files are closed. This does not terminate C-terp but rather allows the program to be rerun with different parameters and/or different input files.

To exit directly to system level from within a program use i_exit().

fclose -- Close a stream

Synopsis:

```
fclose( stream )
    FILE *stream;
```

Description:

flushes output and closes the stream. This is the companion function to fopen(). It returns -1 if there was an error and 0 otherwise.

feof -- Test end of file

Synopsis:

```
int  feof( stream )
     FILE *stream;
```

Description:

returns true (non-zero) if the stream is at the end of file, otherwise 0.

ferror -- Return error status

Synopsis:

```
int  ferror( stream )
     FILE *stream;
```

Description:

ferror() returns true (non-zero) if an error has been detected on the stream.

fflush -- Flush an output stream

Synopsis:

```
int fflush( stream )
    FILE *stream;
```

Description:

flushes all buffered information to DOS. This protects against sudden death of C-terp.

fgetc -- Get a character from a stream

Synopsis:

```
int  fgetc( stream )
     FILE *stream;
```

Description:

fgetc() returns the next character from the stream. On end-of-file, it returns EOF (#define'd as -1 in stdio.h).

In C-terp, this function is identical to getc().

fgets -- Get string from a file

Synopsis:

```
char  *fgets( buf, len, stream )
    char *buf;
    int len;
    FILE *stream;
```

Description:

reads a string (up to and including the next new-line) from the indicated stream into the buffer buf. The string will be terminated by a null. If a long line is encountered (exceeding len-1 characters) only the first len-1 characters will be read into buf.

The value returned is buf if the read was successful. If unsuccessful such as at the end of a file, the NULL pointer is returned. This allows each line of an input stream to be processed by the relatively simple loop:

```
while( fgets( buf, buflen, stream) )
    {
    .
    .
    .
```

fileno -- Return the stream number

Synopsis:

```
int fileno( stream )
    FILE *stream;
```

Description:

returns the stream number of the indicated stream. As examples, stdin, stdout and stderr have stream numbers of 0, 1 and 2 respectively. The stream numbers may be used in low level calls that do not make use of streams such as read and write.

This is implemented as a macro.

fopen -- Open a file

Synopsis:

```
FILE  *fopen( file_name, mode )
    char *file_name;
    char *mode;
```

Example:

```
fopen( "data.in", "r" )
```

Description:

A file of a given name is opened in a mode specified by the second argument. A stream designator is returned or, in the event of an error, the NULL pointer is returned.

There are three streams that are pre-opened: stdin, stdout and stderr. These streams are declared in stdio.h

The file_name must be the name of a file (full path names are allowed) or one of:

"CON:" The console (keyboard on input, screen on output)

"PRN:" The printer

"AUX:" The auxiliary (usually communication) port

Modes:

"r" (Read) The file is opened for reading. If the file did not previously exist, the NULL pointer is returned.

"w" (Write) The file is opened for writing. If it did not previously exist a new file is created. If it did previously exist, the file is truncated to zero length.

"a" (Append) The file is opened for output but unlike "w" mode it is not truncated. Rather the current write pointer is set to the end of the file.

"...b" (Binary mode) The elipsis is any of the above. Use of this mode implies that the caller has access to all the bits capable of being transmitted to or from the file. In particular the mode suppresses the usual carriage-return/linefeed translation.

"...+" (Both ways) The elipsis is any of the above. The plus indicates that reading and writing are to be done on the same file.

An interpreted program may use _fmode to specify binary mode (although this is not particularly recommended). To do so _fmode must be declared (as opposed to defined) and then explicitly assigned a value. An entry must also be made in the tblxn.c table.

The number of files allowed to be open at any one time is initially limited by DOS to approximately three (in addition to all the standard files such as console, printer, etc.). It is easy enough to raise this limit. The total number of files allowed by DOS is 8. If you want to increase the number allowed by 4, place files = 12 in your CONFIG.SYS file (refer to the section of the DOS manual that talks about Configuring Your System).

fprintf -- Formatted print to a stream

Synopsis:

```
fprintf( stream, format, arguments to be converted )
    FILE   *stream;
    char   *format;
```

Description:

The characters in the provided format are output to the stream except that special action is taken upon the appearance of a '%' character. The '%' character introduces a conversion specifier having the form:

```
% [control-fields] conversion-character
```

where [ ] means 'optional'. This specifier indicates how the arguments which follow the format string are to be converted. The conversion characters are as follows:

Integral Conversion Characters -- for each of these the argument is presumed to be an integer unless an 'l' (letter el) appears in the control field in which case the argument is presumed long. Note that char's and short's are promoted to integer so that these may also be converted.

d	signed decimal
u	unsigned decimal
x	hexadecimal
o	octal

Floating Point Conversion Characters -- for these conversion characters the argument is presumed to be float or double.

e exponential (i.e. scientific) notation -- The number is printed in the form m.nnnnnnExxx. The number of digits to the right of the period is by default 6 but can be overridden by the precision parameter (see a discussion of control fields below).

f floating point notation -- The number is printed in the form mmm.nnnnnn where the number of digits to the left of the

decimal is governed by the numeric value of the argument and the number of digits to the right is by default 6 but can be overridden by the precision field.

g general -- uses format f or e depending on which requires fewer places.

String Conversion Characters

s string -- The argument is the address of a null-terminated string which is copied without conversion. Padding with blanks and/or truncating is governed by optional control fields specified below.

c character -- The argument is a char (promoted to int) which is output without conversion.

Control Fields

The control field items are optional and, if present, should appear in the order in which they appear below.

A minus sign left-adjust the item in the field (this presumes a field width, see below).

A zero pad the field with 0's rather than with blanks.

A field width specifies the minimum field width the item can occupy.

A precision specified by a period followed by an integer. This value normally controls the number of digits to the right of the decimal point (see e, f and g above). It may also be used to truncate a string.

A long flag specified by 'l' (letter el) indicates that the corresponding argument is a long or unsigned long.

fputc -- Output character to a stream

Synopsis:

```
int fputc( c, stream )
    int c;
    FILE  *stream;
```

Description:

outputs character c to the indicated stream. It returns EOF if an error is detected otherwise it returns c. In C-terp, this function is identical to putc().

fputs -- Output string to a stream

Synopsis:

```
fputs( s, stream )
    char *s;
    FILE  *stream;
```

Description:

outputs string s (a null-terminated string of characters) onto the indicated stream. No newline is appended. EOF is returned on error. fputs() is a companion to fgets(). For example:

```
while( fgets( buf, len, stdin ) )
    fputs( buf, stdout );
```

assuming proper declarations, etc., will copy stdin to stdout a line at a time.

fread -- Read data from a stream

Synopsis:

```
fread( ptr, item_size, nitems, stream )
    char *ptr;
    int item_size, nitems;
    FILE  *stream;
```

Description:

fread() will read an array of data from the given stream into a storage region pointed to by ptr. The array consists of nitems each of size item_size. It returns the number of complete items read.

This function is primarily intended to read binary information (see the b mode in fopen).

free -- Free a region of storage

Synopsis:

```
int free( ptr )
    char *ptr;
```

Description:

A region of memory is relinquished to the memory dispenser for reallocation. The pointer must be one previously obtained from calloc() or malloc(). The value returned is 0 if successful and -1 if invalid.

fscanf -- Formatted read from a stream

Synopsis:

```
fscanf( stream, format, pointers to data )
    FILE  *stream;
    char  *format;
```

Description:

fscanf reads information under format control into data areas pointed to by additional arguments following the format. Input comes from the designated stream. As in fprintf, conversion specifications begin with a '%' character and these two facilities share many of the same specifications. As an example:

```
fprintf( stdin, "%d , %d", &n, &m )
```

will read in two integers from stdin and assign them to n and m (presumed declared as int). The comma in the format implies that the two integers must be separated by a comma. White space (blanks, tabs and newline) in the format are ignored and may be inserted for legibility. White space in the file is generally passed over in search of the data item. Its only significance is that it will terminate data items. Thus in the above example, the two data items could be on separate lines or on the same line provided there is exactly one non-whitespace character between them and that character is a comma.

The value returned by fprintf is the number of items successfully matched including a count of one for every non-whitespace character in the format. Thus, if the above example were completely successful, it would return 3. It returns EOF if end-of-file is incountered.

The general form of a conversion specification is:

% [control-fields] conversion-character

where [ ] means 'optional'. The conversion characters are as follows:

Integral Conversion Characters -- for each of these the argument is

presumed to be a pointer to an integer unless an 'l' (letter el) appears in the control field (just before the conversion character) in which case the argument is presumed to be pointer to long. For compatibility with other systems in which short's are different from int's, an 'h' conversion character implies decimal conversion and the argument must be a pointer to a short.

d	signed decimal
x	hexadecimal
o	octal
b	binary

Floating Point Conversion Characters -- for these conversion characters the argument is presumed to be a pointer to float unless an 'l' (letter el) precedes the character in which case the argument must be a pointer to double.

f exponential (i.e. scientific) notation -- The number contains an optional leading sign, a number with an optional decimal point, and an optional exponential field. The exponential field is an 'e' or 'E' followed by an optionally signed exponent.

String Conversion Characters

s string -- The data item is a string of characters containing no white space. The scanner will search forward until a non-whitespace character is found. It will then copy characters into the buffer whose address is provided as argument. Characters will be copied until a white space is encountered which is not copied into the buffer. A null is appended to the string in the buffer. The buffer must have enough room to contain the string plus one white space. A limit can be placed on the amount of data so obtained by specifying a field width (see below).

c character -- The argument is a pointer to char; the next character on the stream (white space or not) is transmitted to the data area pointed to.

Control Fields

The control field items are optional and, if present, should appear in the order in which they appear below.

*	suppress assignment. The converted value is discarded. No argument should be provided.
A field width	specifies the maximum field width the item can occupy. The field width must be an integer constant.
A long flag	specified by 'l' (letter el) indicates that the corresponding argument is a pointer to a long or unsigned long or, in the case of the e or f conversion

characters, to a double.

fseek -- Reposition a stream

Synopsis:

```
long  fseek( stream, offset, type )
          FILE  *stream;
          long offset;
          int type;
```

Description:

fseek() repositions the stream so that the next read or write will start at the indicated position. The desired position is obtained from offset and type in the following manner. Argument type must be one of:

0 position to offset bytes from the start of the file

1 position to offset bytes from the current position

2 position to offset bytes from the end of the file. offset should be nonpositive.

The value returned by fseek() is the new position or -1 if an error occurs. Thus, the call:

```
fseek( stream, 0L, 2 )
```

returns the last position in the file.

The position is the number of bytes from the beginning of the file. For ASCII mode (i.e., non-binary) this may not be the same as the number of characters obtained via putc() or getc() to get to a point in the file because of newline conversion.

An important special case is

```
fseek( stream, 0L, 0 )
```

which serves to rewind the file.

ftell -- Return current position of a stream

Synopsis:

```
long  ftell( stream )
      FILE  *stream;
```

Description:

ftell() returns the current position of a file in a form usable by fseek. The function has fallen into disuse because it is equivalent to fseek(stream, 0L, 1).

fwrite -- Write data to a stream

Synopsis:

```
int fwrite( ptr, item_size, nitems, stream )
    char *ptr;
    int item_size, nitems;
    FILE  *stream;
```

Description:

fwrite() will write item_size * nitems bytes from the memory area pointed to by ptr to the file designated by stream. The stream would normally have been opened with one of the binary modes (see fopen).

The value returned is the number of items (not the number of bytes) written.

This is a companion function to fread().

getc -- Get a character from a stream

Synopsis:

```
int  getc( stream )
     FILE *stream;
```

Description:

getc() returns the next character from stream. On end-of-file, it returns EOF (#define'd as -1 in stdio.h).

getc() is normally implemented as a complex macro when you are compiling. If you are using stdio.h provided with C-terp then getc() will expand into fgetc() under C-terp (using the #ifdef C-terp directive) and will expand into the normal expansion when you compile. The only reason for the difference is that the macro causes a modification of _iob which is out of range as far as the C-terp check() is concerned.

getchar -- Get a character from stdin

Synopsis:

```
int  getchar( )
```

Description:

getchar() will obtain the next character from stdin. It is equivalent to getc(stdin).

i_exit -- exit directly to system

Synopsis:

```
i_exit( n )
    int n;
```

Description:

i_exit(n) will terminate C-terp and return back to DOS with a return code of n. This is an extreme form of exit. In most cases exit() will serve better.

int86x -- Issue a software interrupt

Synopsis:

```
int86x( intno, inregs, outregs, segregs );
    int intno;
    struct {int ax, bx, cx, dx, si, di; }
        *inregs, *outregs;
    struct { int es, cs, ss, ds; }
        *segregs;
```

Description:

This function enables you to issue a software interrupt, and thereby make a request to DOS or the BIOS.

intno is the interrupt number. inregs and outregs are pointers to structures holding general registers. segregs is a pointer to a structure containing segment registers of which only es and ds are significant for this call.

Prior to making the call on int86x, the program should initialize the structures pointed to by inregs and segregs. Just before the interrupt is issued, DS and ES (only!) of the segment registers are loaded from segregs and the general registers are loaded from inregs. After the interrupt is completed, the outregs structure is loaded and can be examined when the return from int86x is made.

To pass a pointer to a DOS function call, one usually has to specify DS:DX. Pointers in C-terp are long pointers. To untangle the segment-offset do the following. Let p be the pointer. Let pp be &p and typed as pointer to int. Then assign pp[0] to DX and pp[1] to DS. A program thus written in C-terp will work in the large model under Lattice but not the small. To get things to run om both models, you should make an inquiry as to the pointer size via the LPTR pre-processor variable and employ intdos() (available in the Lattice library) in the small model

is ... -- Is a character a member of a class

Synopsis:

```
int

isalnum( c )    /* is c alphanumeric? (alphabetic or digit) */
isalpha( c )    /* is c alphabetic?  (one of the 52 letters) */
isdigit( c )    /* is c a digit? */
islower( c )    /* is c a lower case letter? */
isspace( c )    /* is c a white space?  (blank, tab or newline) */
isupper( c )    /* is c an upper case letter? */

        char c;
```

Description:

See above.

lseek -- Seek to a position in the file

Synopsis:

```
long  lseek( fd, offset, base )
      int fd;
      long offset;
      int base;
```

Description:

lseek() will position the file identified by file descriptor fd to a position specified by offset and base. The allowable base codes are:

0	position is offset bytes from beginning of file
1	position is offset bytes from current position
2	position is offset bytes from the end of the file (offset should be non-positive in this case)

lseek() will return the new position.

The position is the number of bytes from the start of the file. For

files being processed in ASCII mode (see open) this will differ slightly from the number of bytes apparently read or written. The reason for this is that carriage-return/line-feed combinations in the file are translated into a single byte of i/o.

malloc -- Allocate memory

Synopsis:

```
char *malloc( n )
    unsigned n;
```

Description:

The function allocates memory of size n bytes and returns a pointer to it. If the amount requested is unavailable, the NULL pointer is returned.

The region may later be freed by the function free().

max -- Find the maximum of two numbers

Synopsis:

```
max( n, m )
```

Description:

max(n,m) will yield the maximum of its two arguments. max() is implemented as a macro (in stdio.h) and for this reason no types are shown for the arguments and no type is shown for the return value. The advantage of implementing as a macro is that the function becomes truly generic working with floating point numbers equally as well as with integers and, even, pointers. The disadvantage is that the arguments should not have side-effects as each argument is referenced twice in the expansion.

movmem -- Move bytes about in memory

Synopsis:

```
movmem( source, dest, count )
    char  *source, *dest;
    unsigned  count;
```

Description:

movmem copies count bytes from the area pointed to by source to the area pointed to by dest.

If the regions overlap the move is done in such a way that there is no ripple effect. That is, it is as if the information were first extracted from source in its entirety and then deposited onto dest.

min -- Find the minimum of two numbers

Synopsis:

```
min( n, m )
```

Description:

min(n,m) will yield the minimum of its two arguments. min() is implemented as a macro (in stdio.h) and for this reason no types are shown for the arguments and no type is shown for the return value. The advantage of implementing as a macro is that the function becomes truly generic working with floating point numbers equally as well as with integers and, even, pointers. The disadvantage is that the arguments should not have side-effects as each argument is referenced twice in the expansion.

open -- Open a file

Synopsis:

```
int  open( file_name, mode )
     char *file_name;
     int  mode;
```

Description:

open will open a file whose name is provided by the first argument as a null-terminated string. It will return a so-called file descriptor which can be used in low-level i/o operations such as read(), write(), lseek() and close(). The file descriptor is a small integer from 0 to some implementation defined limit.

If the file cannot be opened -1 is returned.

The second argument, mode, specifies how the file will be used as follows:

0 ASCII read
1 ASCII write
2 ASCII read-write
0x8000 + any of the above implies binary.

The distinction between binary and ASCII is that binary is a form of I/O in which all bytes in the file are user-visible. In ASCII mode, carriage-return/line-feed combinations are converted to new-line on input and the new-line is reconverted on output.

Lattice C users may employ an include file (fcntl.h) supplied with their compiler. This contains names rather than numbers and these names may be used to specify the mode.

The following special file_name's are supported:

"CON:" the console
"PRN:" the printer
"AUX:" the communications device

printf -- Print to standard output

Synopsis:

```
printf( format, arguments to be converted )
    char *format;
```

Description:

printf() will output information to standard output under format control. printf(format, ...) is equivalent to fprintf(stdout, format, ...). Refer to fprintf() for a discussion of formats and additional arguments.

Standard output is normally directed to the screen. However, it may be redirected by means of a >file construct on the command line.

putc -- Output a character to a stream

Synopsis:

```
int putc( c, stream )
    char  c;
    FILE  *stream;
```

Description:

The character is output to the stream. The stream is a quantity previously returned by fopen() or one of stdout or stderr.

The value returned is c if the put was successful, otherwise EOF.

putc() is normally implemented as a complex macro when you are compiling. If you are using stdio.h provided with C-terp then putc() will expand into fputc() under C-terp (using the #ifdef C-terp directive) and will expand into the normal expansion when you compile. The only reason for the difference is that the macro causes a modification of _iob which is out of range as far as the C-terp check() is concerned.

putchar -- Put a character to standard output

Synopsis:

```
putchar( c )
    char c;
```

Description:

putchar(c) is a convenient abbreviation for putc(c, stdout).

read -- Read characters from a file

Synopsis:

```
int read( fd, buffer, count )
    int fd;
    char *buffer;
    int count;
```

Description:

read() will read up to count characters from the file specified by the file descriptor fd into the buffer whose address is provided by the second argument.

The value returned is the number of characters read. If there are no characters remaining in the file when the call is made, 0 is returned.

read() will return a -1 on error and hence the best control structure to avoid perpetual looping is some variant of:

```
while( read(...) > 0 )
    {
    .
    .
    .
```

rewind -- Rewind a file

Synopsis:

```
rewind( f )
    FILE *f;
```

Description:

rewind(f) will reset file f to its beginning. It is equivalent to fseek(f,0L,0).

The name 'rewind' stems from an earlier day when the principal secondary storage media was magnetic tape which was literally rewound to get to its beginning.

scanf -- Scan fields from the standard input

Synopsis:

```
int scanf( format, arguments )
    char *format;
```

Description:

scanf() reads information from the standard input and assigns values to variables under format control. scanf(format,...) is equivalent to fscanf(stdin,format,...). Refer to the entry under fscanf() for a full discussion of formats and the appropriate arguments.

setmem -- Set memory to a byte value

Synopsis:

```
setmem( p, count, value )
    char *p;
    unsigned count;
    char value;
```

Description:

setmem() will assign the third argument, value, to each byte in the area of storage pointed to by p and extending for count bytes.

sprintf -- Formatted print to a string

Synopsis:

```
sprintf( buffer, format, Arguments )
    char *buffer;
    char *format;
```

Description:

sprintf() is similar to fprintf() except that the information is directed to a buffer of characters rather than to a file. The null character is appended to the characters transmitted. It is the caller's responsibility to ensure that there is sufficient room in the buffer.

This routine allows the user to harness the powerful number-to-string conversion facilities inherent in the printing of numbers.

See fprintf() for a discussion of the allowed format conversions.

sscanf -- Scan fields from a string

Synopsis:

```
int sscanf( s, format, Arguments )
    char *s;
    char *format;
```

Description:

sscanf() is similar in operation to fscanf() except that information is read from a string rather than a file. The routine allows the user to harness the string-to-number conversion facilities inherent in reading numeric information.

sscanf() is especially useful for scanning user input from the console since fscanf and scanf() skip blindly across line boundaries. For example suppose the user is to respond to a prompt with some number followed by a return. If the user types simply a return an error should be

indicated or a default taken. But fscanf() and scanf() will ignore the return and continue to search for a number. The better way to do this is to read the information into a buffer with, for example, fgets() and scan the information with sscanf().

See fscanf() for a complete discussion of format controls.

The value returned is the number of items matched (the same as fscanf).

strcat -- String concatenation

Synopsis:

```
char *strcat( s1, s2 )
     char *s1, *s2;
```

Description:

Appends string s2 to the end of string s1. The null byte at the end of s1 is overwritten and the null byte at the end of s2 is carried along. It is the caller's responsibility to ensure that there are a sufficient number of characters in the area in which s1 lies. The value returned is s1.

strcmp -- String compare

Synopsis:

```
int  strcmp( s1, s2)
      char  *s1, *s2;
```

Description:

strcmp() compares the two (null-terminated) strings and returns 0 if the string s1 is character for character the same as s2, +1 if string s1 is lexically greater than s2, and -1 if string s1 is lexically less than s2. (By 'lexically' we mean dictionary ordering and the ASCII collating sequence.)

strcpy -- String copy

Synopsis:

```
char *strcpy( s1, s2 )
     char *s1, *s2;
```

Description:

strcpy() copies the characters pointed to by s2 to the area pointed to by s1 up to and including the null character. It is the caller's responsibility to make sure that there is enough space in the area pointed to by s1. The value returned is s1.

strlen -- String length

Synopsis:

```
int  strlen( s )
     char  *s;
```

Description:

strlen() returns the length of the null-terminated string s (does not include the null).

strncmp -- String bounded compare

Synopsis:

```
int  strncmp( s1, s2, n )
     char  *s1, *s2;
     int n;
```

Description:

strncmp() is like strcmp() except that a limit of n characters is imposed on the comparison. That is, the character for character comparison between s1 and s2 stops either at the nth character or a null byte whichever comes first.

It returns, as in the case of strcmp(), 0, +1, or -1 depending on whether string s1 is lexically equal to, greater than, or less than string s2.

strncpy -- String bounded copy

Synopsis:

```
char  *strncpy( s1, s2, n )
      char  *s1, s2;
      int n;
```

Description:

Up to n characters from string s2 are copied to s1. The characters are terminated by a null character.

tolower -- Convert character to lower case

Synopsis:

```
int  tolower( c )
     char c;
```

Description:

tolower() returns the argument c unchanged unless c is an upper case letter ('A' - 'Z') in which event it returns the lower case equivalent.

toupper -- Convert character to upper case

Synopsis:

```
int  toupper( c )
     char *c;
```

Description:

toupper() returns the argument c unchanged unless c is a lower case letter ('a' - 'z') in which event the equivalent upper case letter is returned.

trace -- Initiate or suspend tracing.

Synopsis:

```
trace( n )
    int n;
```

Description:

The argument to trace is a collection of OR'ed bits that will selectively turn on (or off) various trace facilities. See the section on DEBUGGING. The bits are:

Bit 1	Statement Tracing
Bit 2	Function Tracing
Bit 4	Add type information to values printed
Bit 010	Assignment tracing
Bit 020	Clause tracing

It returns the most recent value that trace() had been called with. This can be helpful in a recursive environment.

ungetc -- Push back an input character

Synopsis:

```
int ungetc( c, stream)
    char c;
    FILE  *stream;
```

Description:

ungetc() will push its first argument back onto the indicated stream. It will be reobtained with the next getc() or other read call that uses the stream.

For maximum portability, ungetc() only one character from a given stream at a time.

The value returned is c itself or -1 if an error is detected.

write -- Write characters to a file

Synopsis:

```
int write( fd, buffer, count )
    int fd;
    char  *buffer;
    int count;
```

Description:

write() will output count characters from the indicated buffer to the file designated by fd (a value returned from open()). If the file has been opened in ASCII mode, newlines will be converted to carriage-return/newline combinations.

The value returned will be the value of the argument count unless an error occurs.

LINKING TO EXTERNALS

An important feature of C-terp is its ability to interface an interpreted program with C written functions (and hence assembly written functions following C calling conventions), and external data objects appearing in C and assembly programs. Indeed the bulk of the library is extended to interpretive routines through this interface. Moreover the interfacing mechanism is transparent to the interpreted programs.

Linking involves creating, via the DOS (or compatible) linker, a new version of C-terp enhanced by the addition of one or more object modules.

The steps needed to add an external to C-terp are as follows:

1. Obtain the object module (or modules) -- (This step can be omitted if you merely want to access a member of some library not currently accessible). Compile or assemble a module containing the external name producing an object module (for concreteness let us call this myprog.obj). If you are assembling, you may want to refer to the discussion on Assembly Language Interface later in this section. If you are compiling, use the Lattice large memory model (switches -mL and -s).

   The object module(s) may optionally be placed in a library. For the sake of concreteness, again, call the library mylib.lib.

2. Add the entry name(s) to tblxn.c -- Edit the module of C-terp called tblxn.c (table of external names), so that it will include a reference to your external name. This usually amounts to a declaration and an addition to a structure. There are numerous examples within tblxn.c already so this shouldn't be a problem. For example, to add a function alpha(), add the declaration

   ```
   extern alpha();
   ```

   and a structure entry:

   ```
   { "alpha", alpha },
   ```

   The string is the name by which the external will be known to the using program. This need not match the external name but it usually will. Note that the type of alpha() will be taken from the interpreted program that calls alpha(), not from the above declaration.

   The second structure entry above is the address of the item in question. For functions and arrays, just give the name as shown above. For anything else, the address (unary & operator) of the object must be placed here. This may require an explicit assignment. Examples of this are shown in tblxn.c.

3. Compile tblxn.c using the switches -mL and -s producing tblxn.obj

4. Make a new version of C-terp as follows.

```
link  c tblxn myobj, ctnew,, cterp ctlib mylib \lc\l\lcm \lc\l\lc
```

where c.obj, cterp.lib and ctlib.lib are provided on the distribution diskette, myobj.obj and mylib.lib, if present, were produced in step one and the libraries lcm.lib and lc.lib are the libraries provided with the Lattice C compiler for the large memory model. The result will be a module named ctnew.exe (where 'ctnew' derives from the second argument to link; you may substitute any name you want).

5. Reference the external name in a C program that is to be interpreted. Include appropriate declarations (only integer-returning functions do not need a declaration). Then interpret the program with your new version (ctnew) of C-terp.

An example

Assume we wish to link to the sin() routine in the Lattice libary.

1. Step 1 can be omitted since the routine we want to access is already in object form.

2. We add the declaration

```
extern sin();
```

and the structure entry:

```
{ "sin", sin },
```

to tblxn.c. The structure entry need not be placed in alphabetic order. The reason that the existing entries are alphabetized is for ease of lookup.

Note that the sin() function is not typed as a double. This was deliberate. The table only has to contain an address to the function so the type is not important. Moreover, Lattice C initializers must be strictly type compatible. Had we declared sin() as double here, we would have had to insert the address of sin() during execution. This could have been done in xn_init() but why go through the trouble.

3. In compiling tblxn.c we will use the flags -mL and -s. The first of these specifies the Large (or big) memory model and the second specifies certain normalizing assumptions for pointers. C-terp and the Lattice library are compiled with the -s flag.

4. Create a new version of C-terp with the command:

```
link c tblxn, ctnew, ,cterp ctlib \lc\l\lcm  \lc\l\lc
```

5. This step asks us to create a C program using sin(). About the simplest example we can think of is:

```
main()
    {
    double sin();

    printf("The sine of 1 radian is %f\n", sin(1.0) );
    }
```

Note that we need to declare sin() since it returns something other than an integer. Instead of this declaration we could have inserted a #include "math.h" pre-processor directive at the head of the program. math.h is a file supplied with Lattice C that contains such declarations for all math functions.

When you now type ctnew at command level you will see C-terp's main menu. Loading and Running the above program (with the null command line) will produce the display:

The sine of 1 radian is 0.841471

Considerations in Accessing Data

Let us assume we wish to access the global variable errno (used by the Lattice C library to denote an error code which is symbolically represented in the file error.h). We first add a declaration to tblxn.c of the form:

```
extern int errno;
```

The position of this declaration is not important other than that it should be placed before the table xn_table[ ]. Ideally we would now like to add the entry:

```
{ "errno", &errno },
```

to the xn_table but the Lattice compiler will balk at the attempt to initialize a variable that is typed pointer to function with some other pointer. We resort to an indirect method exemplified by the treatment of the _iob array. We assign the desired pointer at run-time (when xn_init() is called).

Accordingly add to xn_table, just after the _iob entry, the lines:

```
#define IX_errno 1
    { "errno" } ,
```

Finally place the line:

```
xn_table[ IX_errno ].xn_ptr = (FUNCTION) &errno;
```

in the initialization routine, xn_init().

Assigning to External Data

The normal use of the variable errno (in the above example) is in accessing information (i.e., reading as opposed to writing). If, however, we were to attempt to assign a value to errno as in:

```
errno = 0;
```

We would raise an error condition becuase we would be assigning outside of the user-allocated data areas (see the discussion under the check() function). There are two ways of handling this. One is by turning checking off with a check(0) call. Indeed, you might want to sandwich the assignment between two check calls as in:

```
n = check(0);
errno = 0;
check(n);
```

where n is a variable allocated for this purpose that restores the state of checking to the most recent state.

Another method is less direct but might be more convenient if there are large numbers of such assignments and you don't wnat to tamper with working code. This method is described under Flags (see below).

Flags

A third field in the structure of which xn_table[ ] is composed is a flags field. The flags are:

```
ALLOC
FREE
REALLOC
```

Each of these flags has to do with telling the check() facility which memory is write-accessible to the program. For example, the entry for malloc is:

```
{ "malloc", malloc, ALLOC } ,
```

If the ALLOC flag were omitted, the function would still work but the check facility would not know that the memory allocated is part of user space.

The ALLOC flag serves both the calloc(n,m) function as well as the malloc(n) function. It multiplies all integer arguments together and assumes the allocatable region is that large.

The REALLOC flag (which derives its name from the realloc function of Unix) operates as follows:

Like the ALLOC flag it assumes the return value is a pointer to a storage area allocated by the callee. Unlike the ALLOC flag, it assumes the length of the storage area is given by the second argument where the first argument is of type pointer.

Assume, for example, the function allow() is defined as:

```
char *allow(p,n)
    char *p;
    unsigned n;
    {
    return p;
    }
```

Suppose further that allow is in an external library accessible to C-terp via the entry:

```
{  "allow", allow, REALLOC } ,
```

Thus you could allow any region of storage to be accessed from within C-terp by calling allow(address,length).

Assembly Language Interface

The assembly language interface is that assumed by the Lattice C compiler. At function call, arguments are pushed onto the stack in reverse order (i.e., the last argument is PUSHed first). Refer to the section on DATA for a discussion of the layout of data values. This is followed by a CALL, which is a NEAR call for the small model and a FAR call for the large model.

You must leave intact the following registers:

CS DS SS BP SP

Return values are placed in registers. Integers are returned in the AX. Longs are returned in AX-BX (BX is the low order word).

Pointers follow the conventions of long. When returned from functions they are placed in the AX-BX registers (AX is the segment and BX is the offset).

Doubles are returned in the four registers AX, BX, CX, DX. The 8087 (and also IEEE) format is used. Here AX is taken as most significant, BX next, CX next, and DX least significant. The first bit of AX is the sign, the next 11 bits are an exponent (excess 1024) and the remaining bits a fraction (the leading 1 bit not explicitly represented).

Restrictions

Traffic is generally one way with the interpreted routines able to call the compiled routines but not vice-versa.

Pointers to functions are compatible in size only between C-terp and compiled objects. That is, if a pointer to an interpreted function is passed into a library member, the member would not be able to invoke the function. It could pass it back out to an interpreted function which could then call the indirected (dereferenced) pointer. Similarly, a pointer to a function generated internally within a compiled procedure could be passed out to an interpreted procedure which could then pass it into some other procedure which could then

make the call but the interpreted procedure itself could not make the call. All this is because C-terp code is a token sequence and not machine language.

ERROR MESSAGES

Some error messages have an associated error number. By looking up the number in the list below you can obtain additional information about the cause of the error.

In general errors in the range 1-99 are compiler-detected errors. Errors in the range 100-199 are interpreter detected errors. Errors in the range 200-299 are C-terp errors. Within each range, errors in the upper quarter (at or above 75, 175, or 275) are such that they cannot be pinpointed to a particular line of user source. The others can be.

Compiler-errors that can be pin-pointed will direct the cursor to the particular token in the source code that is the most likely source of the error. Be aware however that this can be a crude indicator if the source code contains a macro (a #define symbol) which was expanded (and possibly reexpanded) into many different tokens. In this case the pointer will merely point to the macro identifier.

Run-time errors cannot be localized to any greater precision that the line of the statement in which the error occurred. Thus if a statement extends over several lines this may not yield a very good indicator. Note, however, that in a compound statement such as the if statement we have statements embedded within statements. It will always be the inner more statement that is designated.

1 Unclosed Comment -- End of file was reached with an open comment still unclosed.

2 Unclosed Quote -- An end of line was reached and a matching quote character (single or double) to an earlier quote character on the same line was not found.

3 #else without a #if -- A #else was encountered not in the scope of a #if or #ifdef or #ifndef.

4 Too many #if levels -- An internal limit was reached on the level of nesting of #if's (including #ifdef's and #ifndef's).

5 Too many #endif's -- An #endif was encountered not in the scope of a #if or #ifdef or #ifndef.

6 Stack Overflow -- One of the built-in non-extendable stacks has been overextended. The possibilities are too many nested #if's, #includes's, static blocks (bounded by braces) or #define replacements. The most likely cause is a #define loop.

8 Unclosed #if -- A #if (or #ifdef or #ifndef) was encountered without a

corresponding #endif.

12 Need < or " -- After a #include is detected a file specification of the form <file-name> or "file-name" is expected.

13 Bad type -- A type adjective such as 'long' 'unsigned' etc cannot be applied to the type which follows.

16 Unrecognized name -- A # directive is not followed by a recognizable word.

17 Unrecognized name -- A non-parameter is being declared where only parameters should be.

19 Useless Declaration -- A type by itself without an associated variable and the type was not a structure and not a union.

20 Illegal use of = -- A function declaration was followed by an = sign.

21 Expecting left brace -- An initializer for an indefinite size array must begin with a left brace.

22 Illegal operator -- A unary operator was found following an operand and the operator is not a post operator.

23 Expecting colon -- A ? operator was encountered but this was not followed by a : as was expected.

24 Expecting an expression -- An operator was found at the start of an expression but it was not a unary operator.

25 Illegal constant -- Too many characters were encountered in a character constant (a constant bounded by ' marks).

26 Expecting an expression -- An expression was not found where one was expected.

27 Illegal constant -- A constant expression contains a pointer in a context that expects an integer.

28 Redefinition -- The identifier preceding a colon was previously declared as not being a label.

29 Redefinition -- Either a #define symbol is being redefined or a variable is being redeclared or a label is appearing twice.

30 Illegal constant -- a constant expression contains too many external pointers.

31 Bad field size -- The field length provided was too long. The maximum length of a field in this implementation is 16 bits.

32 Bad field size -- The length of a field was given as non-positive, (0 or negative).

33 Illegal constant -- the address of an auto variable cannot be used as a constant.

34 Excessive size -- The maximum size of a structure was exceeded (the limit is at or near 64K bytes).

35 Excessive size -- The total static storage size was excessive. The limit is at or near 64K bytes.

36 Excessive size -- The limit on the total amount of automatic storage was exceeded. The limit is at or near 64K bytes.

37 Repeated include file -- The file whose inclusion within a module is being attempted has already been included in this compilation.

39 Exceeded available memory -- The demands placed by the compiler on the memory allocator could not be met. It is best to save and quit at this point.

40 Undeclared identifier -- An identifier was encountered within an expression that had not previously been declared and was not followed by a left parenthesis.

41 Break -- A Ctrl Break was entered at the keyboard.

42 Expecting a statement -- A statement was expected but a token was encountered that could not possibly begin a statement.

43 Vacuous type -- A vacuous type was found such as an array with no bounds or a structure with no members in a context that expected substance.

44 Need a switch -- A case or default statement occurred outside a switch.

45 Bad use of register -- A variable is declared as a register but its type is inconsistent with it being a register such as a function.

101 Bad type -- Expected an arithmetic object.

102 Bad type -- Expected an integral object (integer or long or something convertable to either of these types).

103 Bad type -- cannot assign to the data object because the type of the object is either struct, array or some other type inconsistent with assignment.

104 Bad type -- The context requires a pointer.

105 Need member -- The context requires a member of a structure or union.

106 Need structure or union -- The context requires a structure or union (on the left side of a -> or . operator).

107 Out of bounds assignment -- An attempt was made to assign information beyond the normal data limits of a program. If this is not an error then check() should be employed.

108 Can't assign -- An attempt was made to assign a structure or union.

109 Bad type -- Attempt to compare (via one of the four operators: >, >=, <, or <=) two quantities that were not both arithmetic and that were not both pointers.

110 Bad type -- Operands to a shift operator were not both integral.

111 Undeclared identifier -- An identifier was encountered at run-time that has no associated definition.

113 Non-conforming pointers -- to compare or subtract two pointers they must be the same type.

114 Bad type -- A pointer was compared for equality against a non-integral value.

115 Illegal array reference -- In an expression of the form x[y], x is not an array or pointer.

116 Unable to convert -- A request was made to convert a data object to a non-scalar type, i.e., to a type that is not arithmetic and not a pointer.

117 Expecting a variable -- The context requires a variable where only a value is given. The context might be the left hand side of an = operator or the left hand side of a dot operator etc.

118 Field too small -- The value being assigned into a field was too large to fit into the field. This represents a possible loss of information. To correct, mask out the undesired information with the & operator.

119 Stack Overflow -- There was an overflow in the main expression stack. Some expression is too deeply nested or recursion extends too far or some combination of the two exists.

120 Unable to convert -- An actual argument of a function call had a type that was inconsistent with the type of a formal parameter. For example, an integer was passed to a function that expected a long or vice versa, etc. In many of these cases the interpreter could correct the situation on the spot but compilers cannot and for compatibility the situation should be corrected.

121 Argument list too long -- The argument list of an external call was too long.

122 Stack overflow -- The function call depth was excessive.

123 Not a label -- The target of a goto statement is not a label.

124 Illegal function reference -- In an expression of the form f(arguments), f

was found to be not a function.

125 Bad type -- It was impossible to obtain the size of the object provided to sizeof().

126 Erroneous I/O pointer -- The FILE pointer was in error.

127 Bad type -- An external function has a type that is not scalar. That is, it is neither arithmetic nor pointer but some compound aggregate or function which may not be passed via a function call. A pointer to such a type would be allowed.

128 Break -- A Ctrl Break was entered at the keyboard.

129 Can't free -- An attempt was made to free a block of storage that did not appear to be one that was allocated.

130 Exceeded available memory -- There was no more main memory available to carry out intended tasks.

175 No main() -- Of all the modules compiled none of them had an external function named main(). One is needed to know where to start the program.

176 Exceeded available memory -- There is no main memory space left to carry out intended functions.

177 Exceeded available memory -- More virtual storage was requested than was available.

178 Excessive size -- The combined static storage areas exceeded the allowed limit of 0xFFE0.

179 Bad file -- An input file was not properly formatted. It may contain a line longer than 255 characters or an unusual line termination sequence or no line termination sequence on the last line.

200-299 Some inconsistency or contradiction was discovered in the C-terp system. This may or may not be the result of a user error. This inconsistency should be brought to the attention of Gimpel Software.

DESIGN NOTES

Why was C-terp written and why did it turn out the way it did? This section is intended to supply at least a partial answer to such questions.

The raison d'etre for C-terp is fast compilation, to eliminate the agony of waiting. To obtain fast compilation, something had to be given up and this was fast execution. This is tolerable as long as C-terp remains compatible with the user's normal compiler. That is, the user must be able to switch back and forth between C-terp and his compiler painlessly.

Since compiler compatibility was important, the ability to link to external functions (and other data objects) was also important. Otherwise, the user would have to have on hand a source written version of all routines used from the compiler-provided library. This may be impractical if the library is proprietary or if the routines are written in assembly language.

Compiler compatibility was important from another standpoint. The language supported by C-terp could not depart from standard C either by omission or addition. It is obvious why a feature should not be omitted; addition would be tolerable but useless if the program is to be eventually compiled.

One early design decision was whether or not to write C-terp in C. Since a lot of code had to be squeezed into 64K (for the small model) and since it had to operate very quickly a not unreasonable choice would have been assembly language. For portability and ease of writing, however, C was chosen. Time critical portions were rewritten in assembly language. The tool used to determine which portions of C-terp were time critical was C-prof, an execution profiler available from Gimpel Software.

One major design decision was whether to compile into tokens (symbol table pointers) or to compile into hard addresses (offsets into the stack and the like). The former is the route taken by such languages as Lisp, Snobol, etc. and the latter is the route taken by Pascal P code compilers. Compiling into tokens was chosen because it was (a) faster and (b) would provide better tracing and diagnostic facilities. It means, however, that type-checking is done at run-time which further slows execution.

Initially it was felt that a bare bones interpreter would be all that would be needed. Facilities such as editing, etc. could be left to the normal editor employed by the user. But to get into and out of editing slows the user down too much. Also, if multiple modules are supported, then to get the full benefits of the multiple module support either a kind of object module would have to be sustained on disk or everything retained in primary memory. If the object module approach were chosen, then much of the advantage of compilation speed would be lost and hence it was decided to retain all information in main memory. This meant caring and looking after a collection of modules rather than just one and it implied linking them together in preparation for a run. It also meant having an editor around.

Initially the editor was to be kept very simple, such as, for example, the BASIC editor. But lacking such humdrum facilities as a context search proved to be a decided disadvantage. Orchestrating the activities of several modules seems to demand other facilities without which one would need to escape too

often to the operating system. Principal among these are inter-file copying and moving and the global search facility to find all instances of a particular string.

The design of the editor follows firstly the special characters on the IBM keyboard (cursor movements and the like). Secondly, it follows somewhat the IBM personal editor since many users of C-terp are likely to have experience using that editor. Thirdly, the editor was oriented to the editing of programs. The auto-indent feature, the uniform tab spacing, splitting and joining, etc. are facilities that are quite useful for programming in a structured (i.e., indented) language.

A fourth driving factor was convenience. There is an advantage to writing a basically simple editor and that is you can avoid most of the multiple control character sequences that a larger editor requires. Things that one does often such as looking for context (Alt L) or resuming the search (Alt N) should be triggered by a single character. Most of the Alt-Letter combinations were chosen so that the Letter occurs on the right side of the keyboard (i.e. the side opposite from the Alt key). This avoids many of the awful finger-twisting exercises involved in typing Alt (or Ctrl) followed by a key on the left.

The library that comes with C-terp is a good bit richer than was originally envisioned. It was originally expected that there would hardly be any at all. Since C-terp could link to the user's normal (compiler-provided) library why not let him do that and be done with it?

Aside from the fact that it is unpleasant to pay for something that is not reasonably self-contained there were a few technical problems to that approach. Most of these could be corrected by adding certain routines to standard C libraries. One problem is the file closing problem. A function was needed to close all files so that, for example, the "Hello World" program would print out before returning back to the main menu rather than after the user quits from C-terp, as actually happened early in the implementation. Another problem is space allocation. If the user is using malloc and free to allocate and/or release storage, C-terp must be able to at least restore back to a known state when execution is over. This sort of thing is done in Pascal with the Mark and Release procedures but no counterpart exists in C. Another problem is exit(). Many C programs call upon this routine to terminate execution. The way most libraries implement this function is to return directly back to the operating system rather than simulate a return from main(). In the C-terp environment, the required functionality is to return back to C-terp's main menu as a direct return to the operating system would lose the user's state.

In view of these problems, a compiler library cannot be used in its unvarnished state but must be modified or, at least, augmented with appropriate additional calls.

Another issue was the wild pointer problem. C, as a language, supports the free and open use of pointers and, equivalently, unhindered subscripting of arrays. For efficency considerations this is reasonable. For a debugging interpreter, time is not such a pressing concern and it would be reasonable to check pointers and subscripted arrays at least for assignment into data areas. Initially the pointer was implemented as a compound object containing a block

number and an offset into the block. A block could be looked up by index number to determine where it actually lay in memory and how big it was. This was ultimately dropped because of the higher priority of being compatible with a C-written library. Specifically, if a data structure were being passed to a library routine (presumably via pointer) all the data in the structure would have to be at least the same size and for true compatibility mean the same thing. Thus C-terp could not assume a closed universe. But some degree of pointer protection is highly desirable if only to protect C-terp itself. For this reason a gross sanity check is made on the target address when information is assigned at the program's request. This check can be turned off with the check() function.

With all of these components working together the result appears to be more than the sum of the parts. With a fairly large multiple-module application one gets the feeling of striding above the various component modules and include files, rather than working underneath. Since compilation is fast it is common to call for compilation immediately after entering each function while that function is fresh in one's mind rather than waiting to get a whole jumble of messages to disentangle later on. Also, it makes sense to provide just one message about one error that the user can respond to immediately rather than let the floodgates open to an avalanche of error messages. The latter is more appropriate to a batch setting.

The intermodule (global) searching seems to be more important than it was originally thought to be. The number of times it is necessary to inquire about the instances in which a function name or global variable is used seems to be much greater than I would have guessed from experience in a world in which it is time-consuming or inconvenient to obtain such information.

RECOMMENDED READING

- Kernighan, B. and D. Ritchie,
The C Programming Language, Englewood Cliffs,
Prentice Hall, 1978.

- Plum, Thomas,
Learning to Program in C,
Cadiff, NJ, Plum Hall, 1983.

- Purdum, Jack,
C Programming Guide,
Indianapolis, Que Corp., 1983.

- Purdum, Jack,
C Programmers Library,
Indianapolis, Que Corp., 1984.

- Harbison, S.P. and G.L. Steele, Jr.
A C Reference Manual,
Prentice Hall, Englewood Cliffs, N.J., 1984.

Timely Notes and Supplementary Information for

C-terp

Version L 2.09
(Lattice Variant)

Additional Features

\x escape

String and character constants may employ the \x construct where \x must be followed by two hexadecimal digits comprising the character. Thus '\x27' is the ESC character. Such constants are not in K&R but are in many compilers and will be part of the new C standard.

Command line options

The flag -c on the command line allows nested comments.

The options -p<path> where <path> is a semi-colon separated list of directories (just as in the MS DOS PATH command) provides for an automatic search facility should a file not be found in the current directory. (See also the Options command).

Main Menu commands

Two commands have been added to the main menu (see THE MAIN MENU in the manual):

System — When 's' is selected on the main menu you will be placed at the DOS level. You should receive the familiar DOS prompt. To this prompt you may respond with any DOS command. (This includes internal commands such as DIR, external commands such as MASM, and batch commands). To return from DOS type EXIT as a command.

Options — If you select 'o' from the main menu you will be presented with a sub-menu of options. These are:

c — Allow nested comments.

p — You are prompted for a path which is a list of directories separated by semi-colons in which to

search for a file that is not found in the current directory. This applies to any file being read for whatever reason (whether #include file or explicit request). The path here is similar to the path argument provided to the MS DOS PATH command. For example:

```
a: ; \include
```

will, if a file is not found in the current directory, result in a search of the device a: and, if not found there, the directory \include. The prompt will include a bracketed old path which is selected if you merely type return. To effectively remove the path enter a semi-colon when prompted for the path.

To set a path in a BATCH file (see the section BATCH MODE) use:

```
o
p path-desired
```

Interactive Debug commands

In the chapter entitled DEBUGGING, section Interactive Debugging, there are two additional commands, viz. Flip and Side-step:

F Flip to the previous screen. The screen on entry to debug is saved. This command allows viewing of this screen without returning to execution. Typing an additional key restores the screen to interactive debug. The interrupted screen is restored prior to continuing the execution. If Flip does not change the screen it is because insufficient memory was found.

S/s Side-step is like Next-step except that it skips over any function calls. That is, it does not stop if the call level is higher than the current one. This is very often the more useful of the two single stepping commands. Like the Next-step command the initial letter of the side-step command is case sensitive. A lower case letter causes no screen flipping, an upper case letter causes the screen to flip. (See below.)

There is also a change to the Next step command.

N ... (continued from manual) This command (as well as Side-step) is case sensitive. If a lower case 'n' is typed the next step is taken with no interchange of the

screens; if an upper case 'N' is typed the next step is taken in such a way that the screen is first flipped to the execution screen, the next statement is executed, and the screen is reflipped back to the debugging screen. The 'N' is advantageous if you are producing tracing information and/or printing on the execution screen while single-stepping. The 'n' is advantageous if the program is not producing screen output because the screen flipping in this case is unnecessary and can be a little jarring to the senses.

Numeric prefixes -- The commands Next-step, Side-step and Continue can be prefixed by a numeric prefix which we represent by <NUM>. These have the following meanings:

<NUM>n Take NUM steps.

<NUM>s Take NUM side-ways steps.

<NUM>c Continue on to line number NUM (of the currently displayed file).

Return-value suppression -- Also in the section Interactive Debugging, the expression following the Display command may optionally be followed by a semi-colon (;) to suppress the printout of the value of the expression. Accordingly, the calls to printf() and dumpv() which appear in the examples of expressions in interactive debugging (page 16) should, more properly, have been followed by semi-colons. We suggest that you mark your document accordingly.

Additional Error Messages

---

18 Illegal Constant -- The sequence \x was not followed by two hexadecimal digits.

131 Unrecognized Name -- The name (actually the one character mnemonic) of a command was not recogized in a batch file.

132 Bad type -- A pointer was expected by an external function.

Pointer checking revised

Loin cloth checking -- check(0) no longer completely eliminates pointer checks. (See function check() in the chapter entitled THE LIBRARY.) If you attempt to assign a value to a region of storage, the segment portion of the pointer is

checked for 0. Said another way, you are no longer running naked with check(0).

The advantage of this checking is that a frequent source of error is the erroneous promotion of int's to pointers which leaves the segment 0. There is almost no disadvantage since there is never any legitimate use of a zero-segment pointer. It does mean that you cannot deposit information directly into the 0 paragraph (the least 16 bytes) but if you want to do this you should be using DOSCALL 25 (that's interrupt 21H, AH = 25H).

Assigning to External Data -- The discussion entitled "Assigning to External Data" in the section "LINKING TO EXTERNALS" indicates that check(0) must be set to assign values to external data. Starting with version 2.06 of C-terp, this is no longer necessary. Such regions of storage are now automatically identified as being accessible for assignment.

New tblxn flags -- For the purpose of automatically checking pointers passed to external functions (such as fprintf, etc., etc.) a set of new flags have been defined. These are in addition to the flags already mentioned in the subsection "Flags" in section "LINKING TO EXTERNALS".

P1 P2 P3 P4 P5

indicate respectively that the first, second, third, fourth and fifth arguments of the function are pointers that should undergo the prevailing pointer check.

P5TAR

indicates that all arguments from the last pointer to be checked onwards are to be pointers to be checked via the prevailing pointer check.

Thus: P2 ! P5 will check the 2nd and 5th arguments. P2 ! P5TAR checks arguments 2, 3, 4, etc. It can be used with a function like scanf() to check all arguments from the 2nd on for being valid pointers. Also, P1 ! P3 ! P5TAR will check the first, third, fourth, fifth and all subsequent arguments.

---

tblxn.c Augmenter

By popular demand we are now providing a utility routine in the form of an executable (augtblxn.exe) and source (augtblxn.c) to augment the file tblxn.c in a painless (even joyful) manner. This is useful if you are adding many external names to the library. See the discussion on pp. 52-53 of the manual (steps

2) in the section LINKING TO EXTERNALS.

To find out how to use augtblxn type

augtblxn

at command level (DOS level).

Lattice version dependencies

The file c.obj furnished on the distribution diskette corresponds with version 2.14 of the Lattice compiler. If you are using the Lattice 2.15 compiler please use c215.obj in place of c.obj when you are making a new version of C-terp (see Section LINKING TO EXTERNALS, p. 52).

For the purpose of showing how c.asm is modified for C-terp, the source for c.asm has been included under the names c214.asm and c215.asm.

Alignment

On page 15 of the manual (Alignment subsection of the DATA section) there is a discussion of alignment. This discussion is erroneous in almost every respect. The paragraph should read as follows:

Data in C-terp (Lattice variant) is aligned in a manner consistent with the Lattice compiler when compiling without the -w flag (the -w flag requests the Lattice compiler to force arithmetic items to align on an even byte boundary). That is, in C-terp, pointers, unions and structs are aligned on a word boundary and unions and structs are padded to an even number of bytes. There is no attempt to align any other item on any particular boundary.